Programming Techniques for Moving Scientific Simulation Codes to Roadrunner

Paul Woodward,
Jagan Jayaraj, Pei-Hung Lin, and David Porter

Laboratory for Computational Science & Engineering
University of Minnesota

Roadrunner Tutorial #5
March 12, 2008

Special Features of PPM Numerical Algorithm:

- ✓ 1-D passes result in difference stencil that is wide in the direction of the pass.
  This suggests strategy for extreme pipelining.
- ✓ Global communication needed only to set the time step value, and out of date information can be used.
  This allows purely local communication and eliminates all need for barrier synchronization.
- ✓ Huge amount of computation performed in order to deliver high accuracy.
  This eases the memory bandwidth requirement.

So How Do I Do That?

- ✓ 1-D passes can be used even in truly multi-D numerical algorithms.
  Evaluate all the X-derivatives in the X-pass.
- ✓ Global communication is highly overrated, but I admit that this view is controversial.
  Even if you must do it, perhaps you can find something to do while it is happening.
- ✓ Huge amount of computation performed.
  Everybody is doing a huge amount of computation, but perhaps not all in the same time step or in the same physics module.

So What is Special About Cell?

- ✓ The challenge of Cell programming is almost entirely derived from its very small on-chip memory (the "local store").
- ✓ The aggregate performance of a Cell CPU is *also* almost entirely derived from the small on-chip memory.
- ✓ Otherwise, the Cell SPE core can be viewed as a scientific computing core. It has all you need and not a single thing more.
  (You really didn't need all that on-chip memory. Really. Believe me.)

Why can't I just recompile my present code?

- ✓ You probably can, but on Cell you probably won't like what you get.
- ✓ Suppose that your code is "blocked for cache."
- ✓ This means that you operate on an on-chip data context that is about 2 MB, and you rely on cache hardware to push out what you no longer need and draw in what you want automatically.
- ✓ On Cell, you have 256 KB, 8 times less space, and nothing is automatic. You want something on chip, you get it. You want it off, you "put" it.

Cell SPE processing paradigm:

- ✓ You explicitly lay out your data workspace.
- ✓ You explicitly fetch data from memory into local arrays.
- ✓ You perform a computation that makes NO references to any off-chip data.
- ✓ You put results into local arrays.
- ✓ You explicitly write these results back to main memory arrays.
- ✓ You must figure out how to REUSE all your local, on-chip arrays. You must semi-continuously empty and refill them.

YOU are in control:

- ✓ Cell treats you like an adult.
- ✓ The system software assumes that you know what you are doing and does not attempt to hide the complexity of the system from you.
- ✓ If indeed you know what you are doing, you will be rewarded with outstanding performance.
- ✓ IBM is building software to address use by casual programmers, but for now, it is reasonable to assume that there are no casual Cell programmers.

This is a stream processing paradigm:

- ✓ Data arrives semi-continuously and asynchronously with respect to all processing.
- ✓ Arriving data is stored in local named arrays which are continuously refilled after they have served their purpose for the data they contain.
- ✓ This same use-and-refill model happens all the way down the line – a modality that demands fully pipelined processing.
- ✓ As results appear at the end of the pipe, they are written back to memory in a return asynchronous stream.

This is exactly what you should have been doing:

- ✓ This stream processing paradigm is optimal for all cache-memory based machines.
- ✓ You should have been doing it for the last 15 years.
- ✓ Don't feel bad, no one else was doing it either.
- ✓ ONE REASON NO ONE WAS DOING THIS IS THAT COMPILERS DO NOT GENERATE THIS SORT OF CODE.
- ✓ Compilers do not generate this type of code because they need your help to do so and are too proud to ask you for it.

Haven't you been doing SOMETHING right?

- ✓ Of course.
- ✓ In this tutorial, we will not dwell on these things, since you already know them.
- ✓ We will therefore gloss over:
  1) Domain decomposition.
  2) MPI message passing.
  3) I/O
     (here we give you the benefit of the doubt).
  4) Writing vectorizable loops.

We will only mention any of the above list to set the context for the hard/new stuff.

General Plan of Computation:

- ✓ Chop up the grid domain into "grid bricks."
- ✓ For simplicity, we will assign one grid brick to each Cell processor.
- ✓ We do 1-D passes in a symmetrized sequence of XYZZYX.
- ✓ In each pass, data is exchanged with only 2 neighbors, in the direction of the NEXT pass.
- ✓ Messages are built as contiguous blocks of data in memory and then dispatched all at once as early as this is possible.
- ✓ Actually having to wait for a message to arrive is considered a programming failure.

General Plan for I/O:

- ✓ A separate process is assigned to do all I/O for a specific set of nodes (1 to 64, depending).
- ✓ These processes can run on the Opterons, which otherwise have nothing to do.
- ✓ Data is compressed for output on the SPEs, and aggregated and potentially reformatted by the I/O processes on the Opterons.
- ✓ This is also true for restart dumps, which make use of the otherwise useless Opteron memory to hold a problem image while it is written out to disk as the code runs forward.

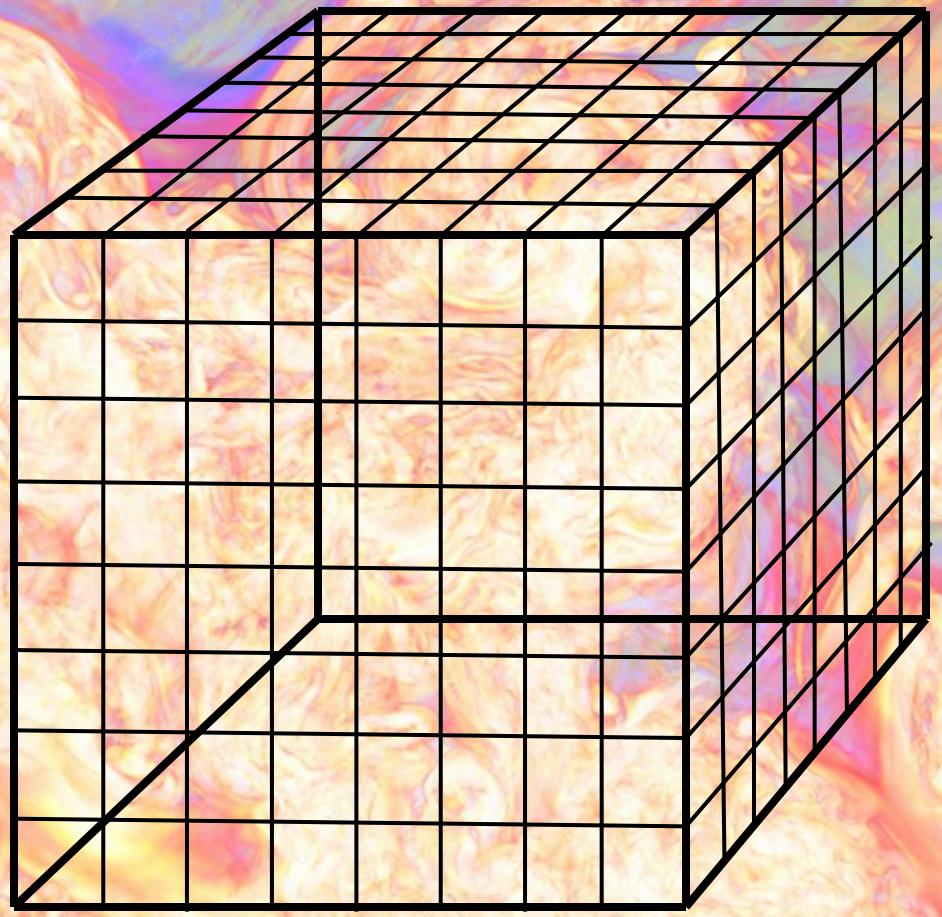
This Plan will Work on Anything:

- ✓ If you design your Roadrunner code carefully, you can run it on every system on earth.
- ✓ But you have to be sure to maintain a standard Fortran version of the Cell SPE code.
- ✓ The Cell processor can be emulated in OpenMP that will run on anything.
- ✓ The Opteron functions will of course run on anything and can be implemented as extra MPI processes.
- ✓ So you can leverage your effort in moving your code to Roadrunner. You will see improvements on every computing system.

Let's Look at the Code for a single Grid Brick:

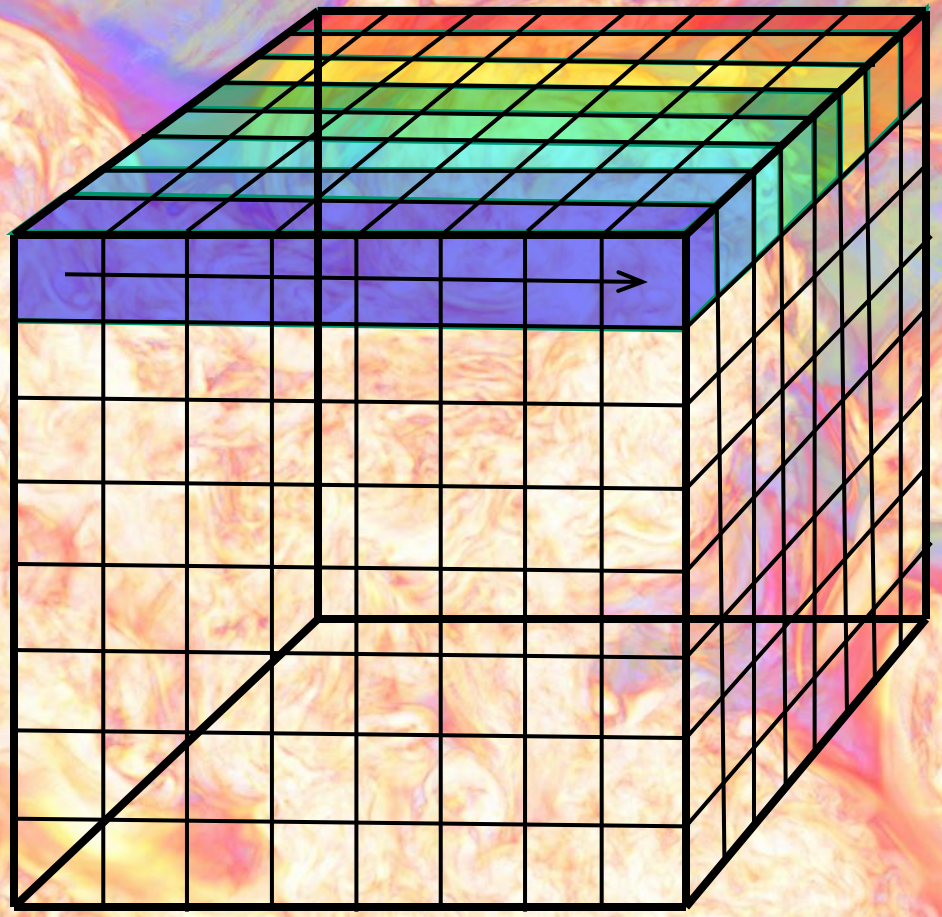
- ✓ This is the code for a single Cell processor.
- ✓ If we make this grid brick small enough, and show that the code will still be efficient, you can believe that the code will scale to a million cores.
- ✓ This would be just 128,000 processors.
- ✓ We take a cubical brick of 32 cells on a side.
- ✓ We will have, with our million cores, a grid for the problem of 40×40×80 bricks, or of only 1280×1280×2560 cells.
- ✓ This is "nothing."

The Biggest Change is the Data Structure.



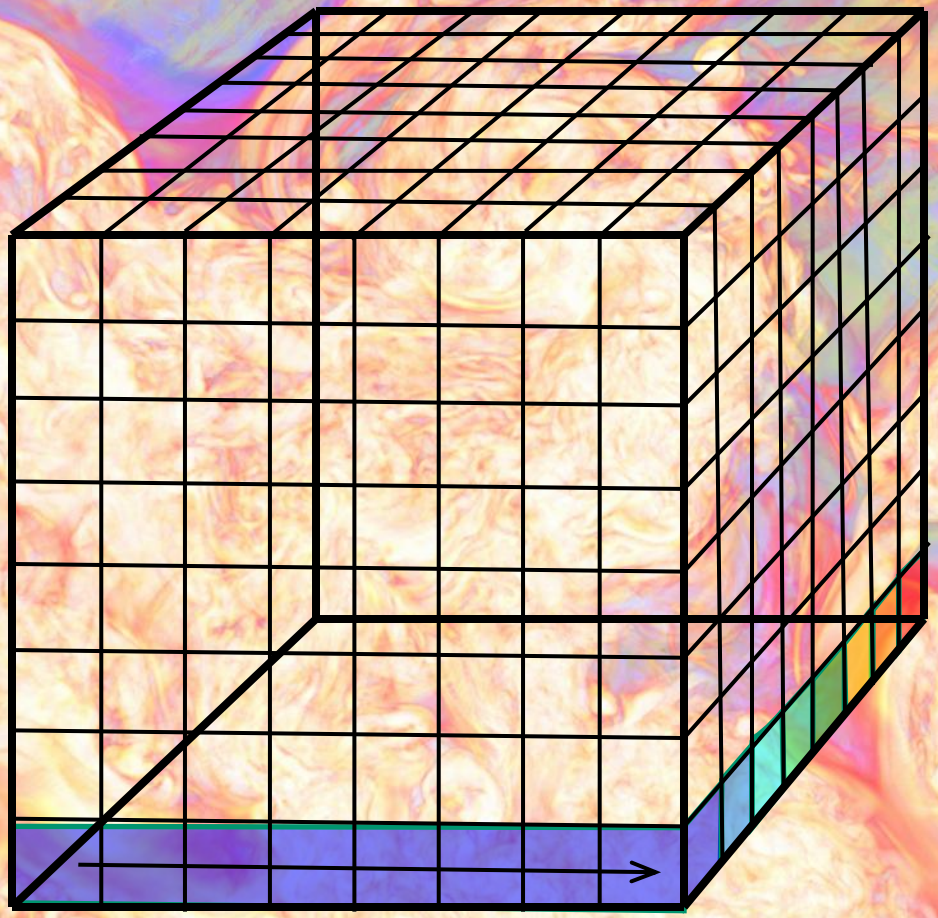
Each grid brick of 32^3 cells is decomposed into 8^3 sugar cubes of 4^3 cells each. On each Cell processor, 8 CPU cores cooperate to update this grid brick, using shared memory to facilitate the process. We will consider an implementation with 2 grid bricks per dual-Cell blade, and 4 per Roadrunner node, because this is a simple and effective approach.

8 cores (different colors) simultaneously update 8 strips of sugar cubes.



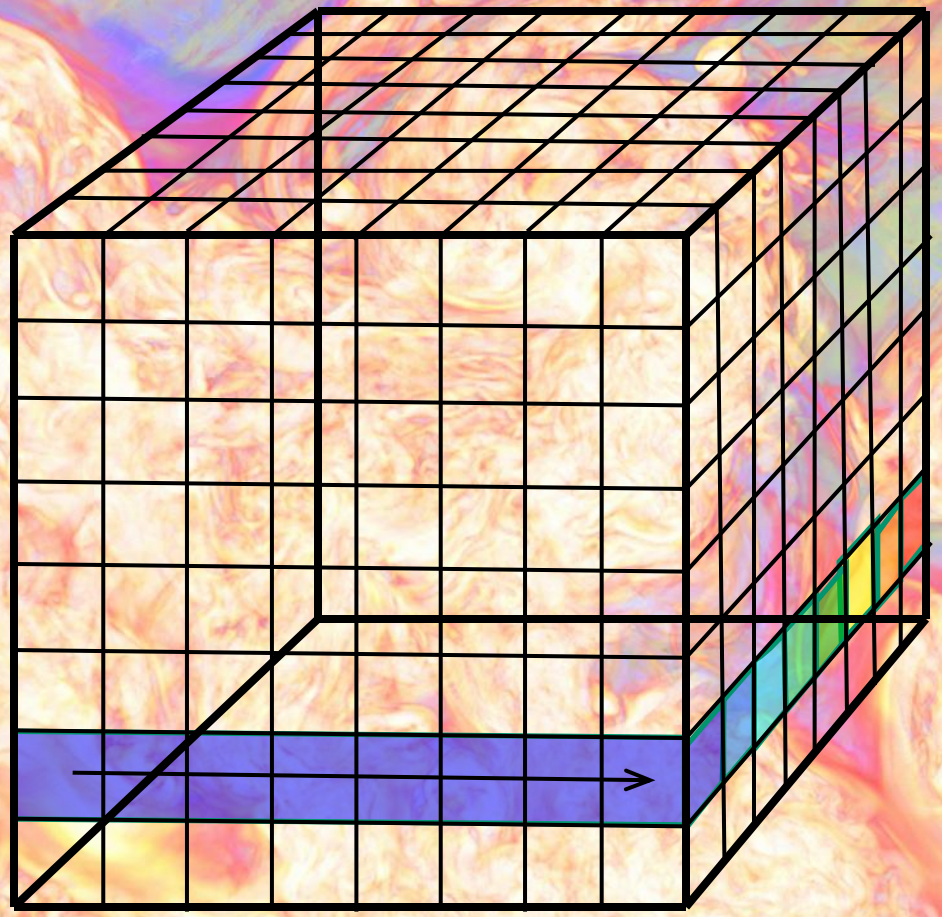
Each CPU core computes, based upon its ID number, which slab of sugar cubes, oriented in the X-Z plane, it will update in the X-pass. The updating of these slabs proceeds in parallel. The top plane is updated first, so that the results can be immediately dispatched as an MPI message for use in updating the brick above this one in the Y-pass.

We update the bottom plane directly after updating the top plane.



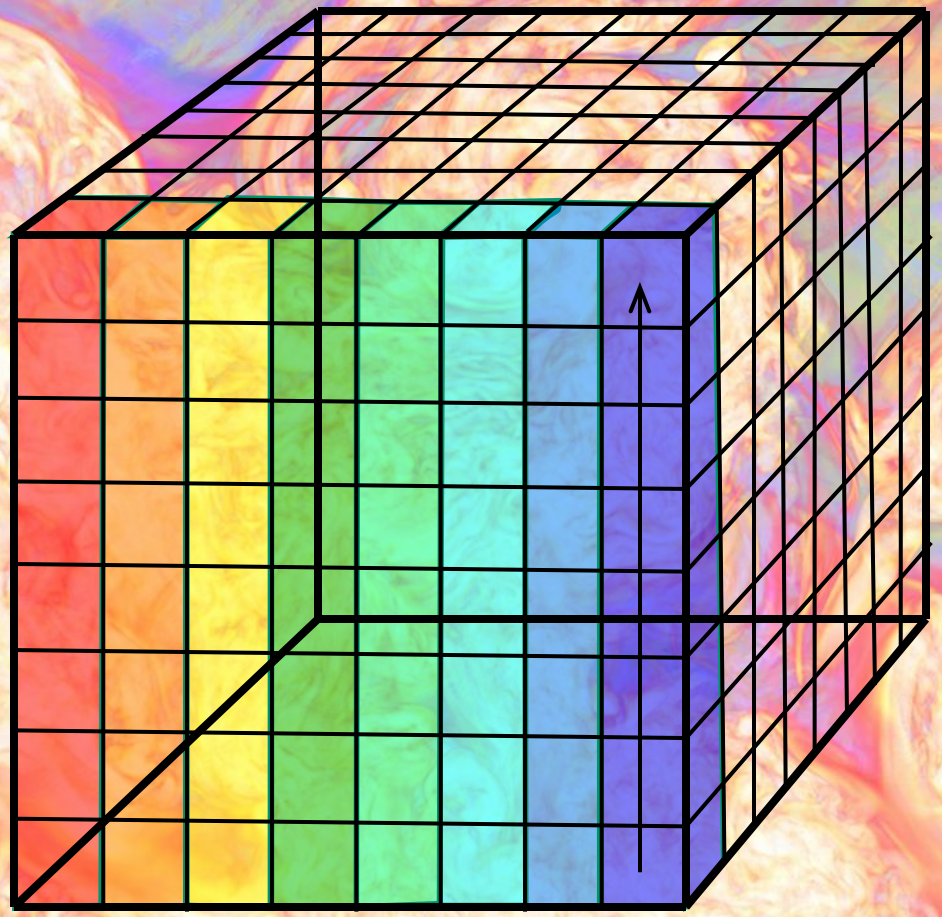
It requires less coordination to wait until both the top and the bottom planes of sugar cubes have been updated before dispatching both MPI messages. The messages are constructed in separate arrays, so that each one is a large, contiguous block of data (a concatenation of sugar cube records) that can be transmitted efficiently via MPI.

We update the grid brick interior while the MPI messages are in transit.



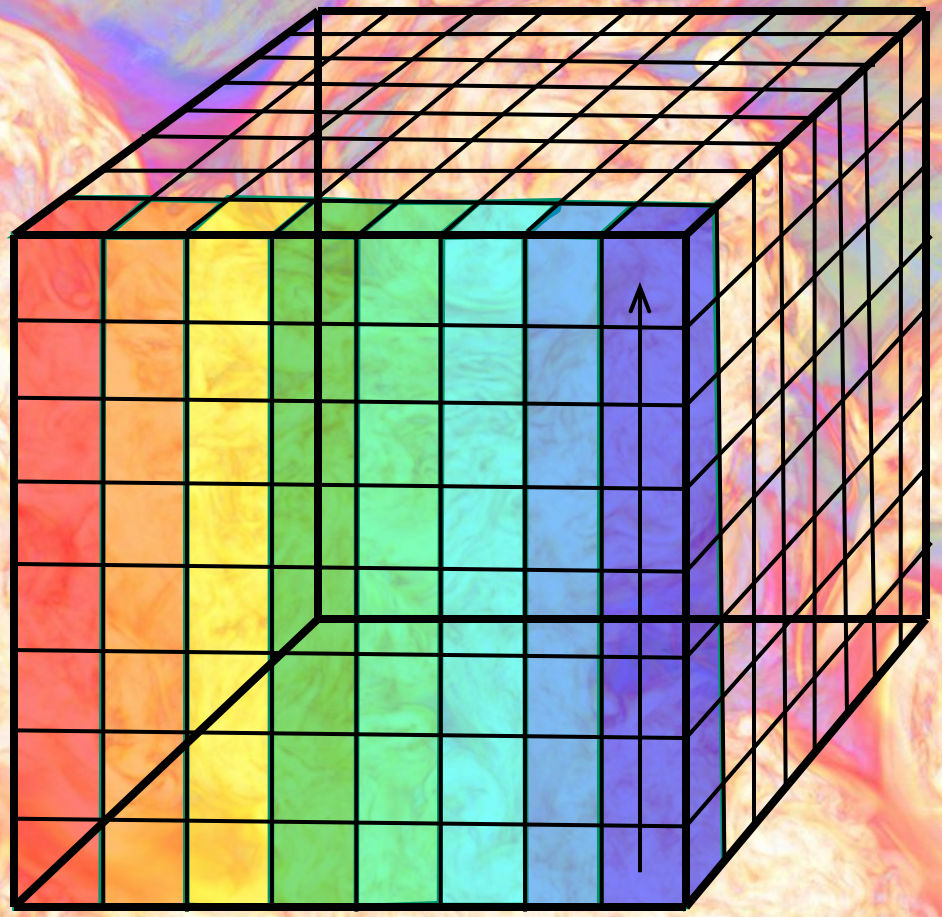
It takes 3 times as long to update the grid brick interior as it does to update the top and bottom planes of sugar cubes. This gives the two MPI messages gobs of time to arrive at their destinations. They will be needed at the outset of the next Y-pass. We can count on them having arrived at that point in the program, but we do insert “waits.”

We begin the Y-pass by updating the near plane of sugar cubes, then the far.

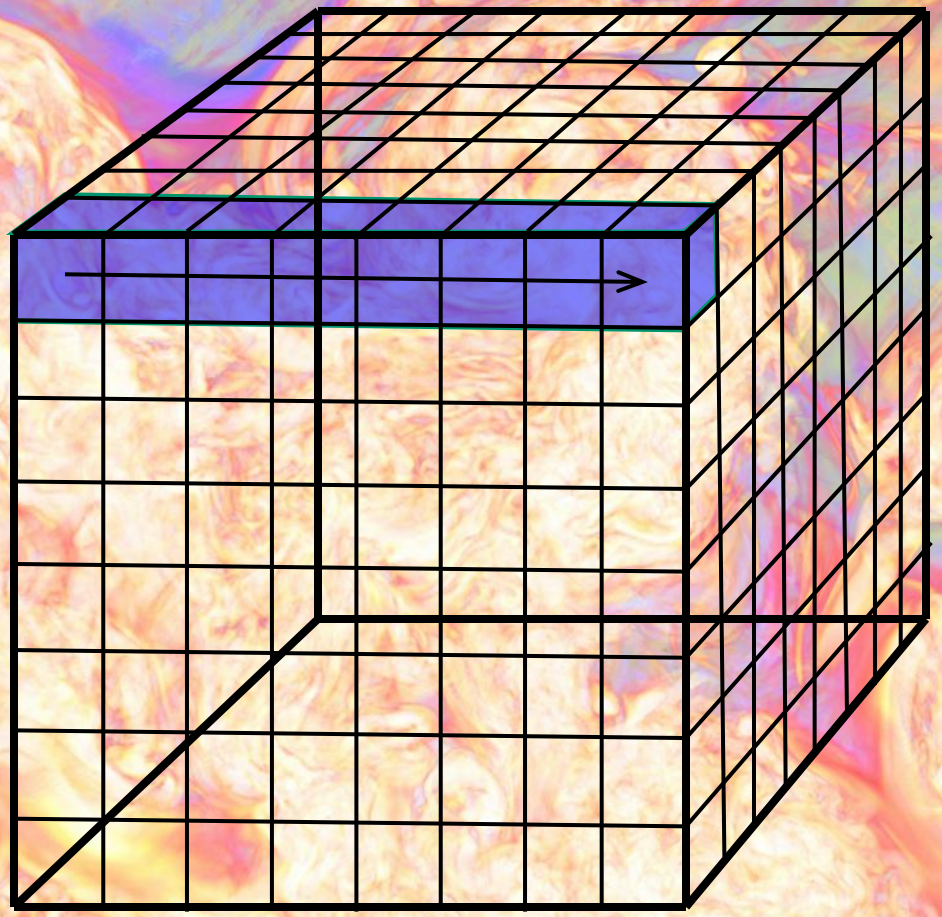


We cannot start the Y-pass until all the interior grid planes are updated in the previous X-pass. For simplicity, we place a barrier synchronization at the end of the full grid brick update for the X-pass, after which we must wait on the arrival of the MPI messages from neighboring nodes in Y. Then we begin with the Y-pass update of the near plane of sugar cubes.

We begin the Y-pass by updating the near plane of sugar cubes, then the far.



Our barrier synchronizing the work of all 8 SPE cores on the same Cell CPU is very inexpensive, since these can communicate in nanoseconds with each other. We wait on MPI messages, but we have NO global barrier synchronization of our many MPI processes. That would be a disaster for performance. And it is completely unnecessary.



Now let's consider the process of updating a single strip of sugar cubes on a single Cell SPE core.

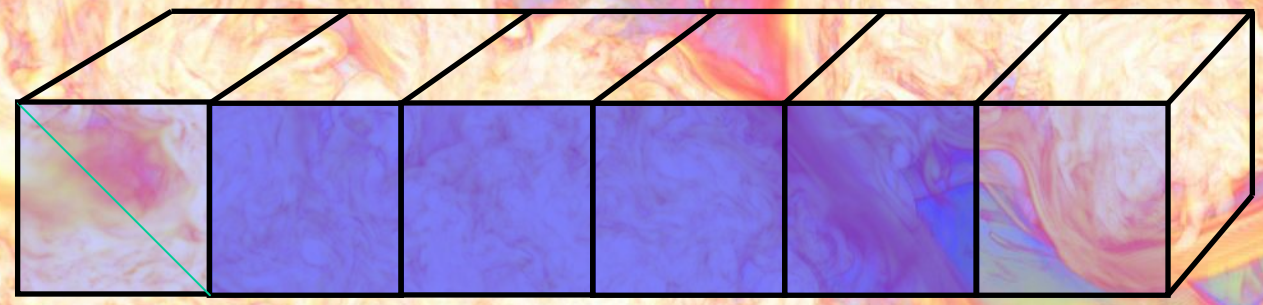
We show just this single strip of sugar cubes on the next slide.

Our single-fluid PPM algorithm for flows of Mach 2 or lower requires one ghost sugar cube at each end of our strip in order to produce updated results in the central 4.

We prefetch one sugar cube while we unpack and operate on the previous one cached in our local store.

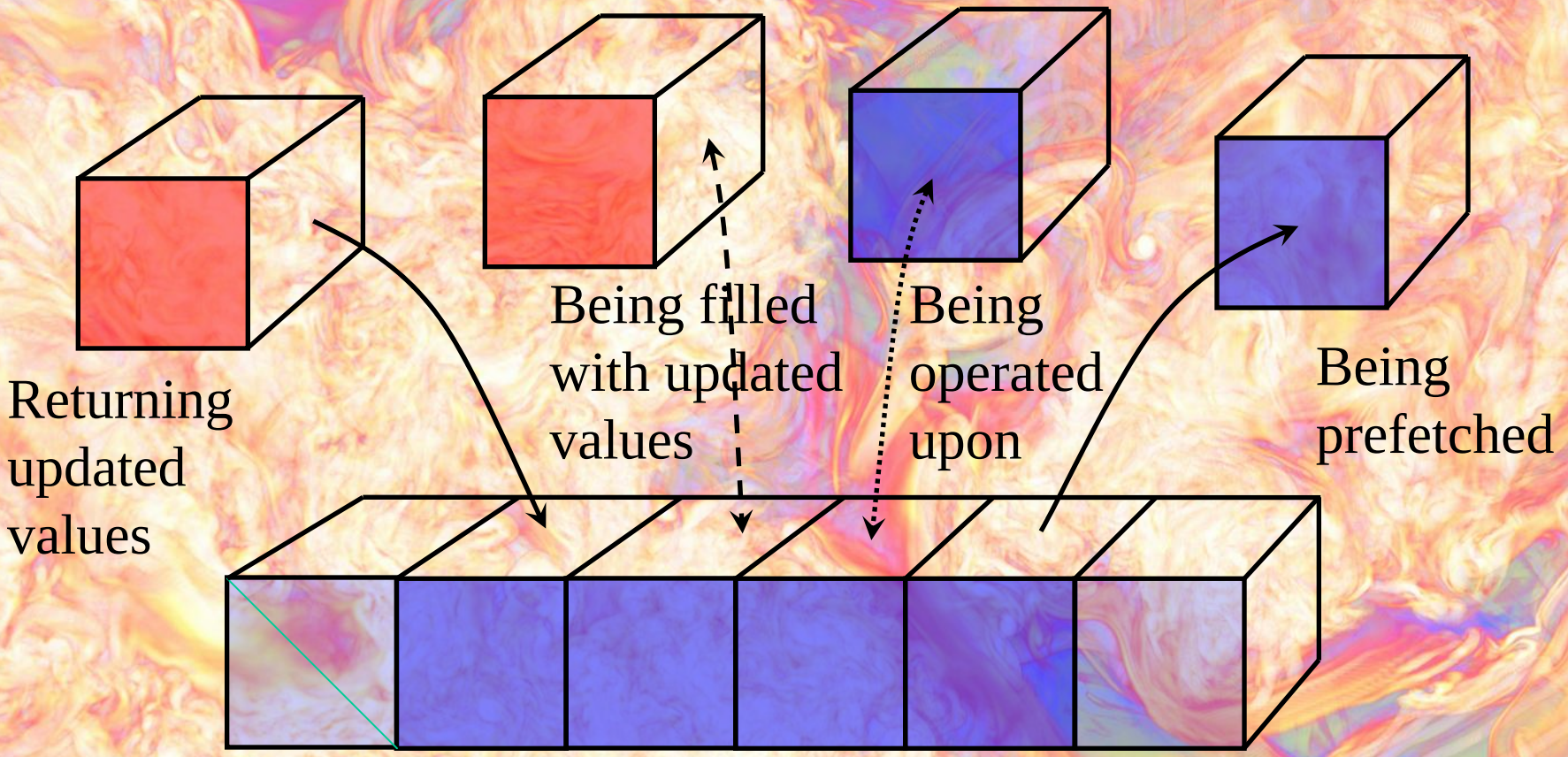
We write back one updated sugar cube while we fill in values in another one cached in our local store.

In preparation for the Y-pass, we transpose the internal contents of each updated sugar cube record before writing it back to the main memory.



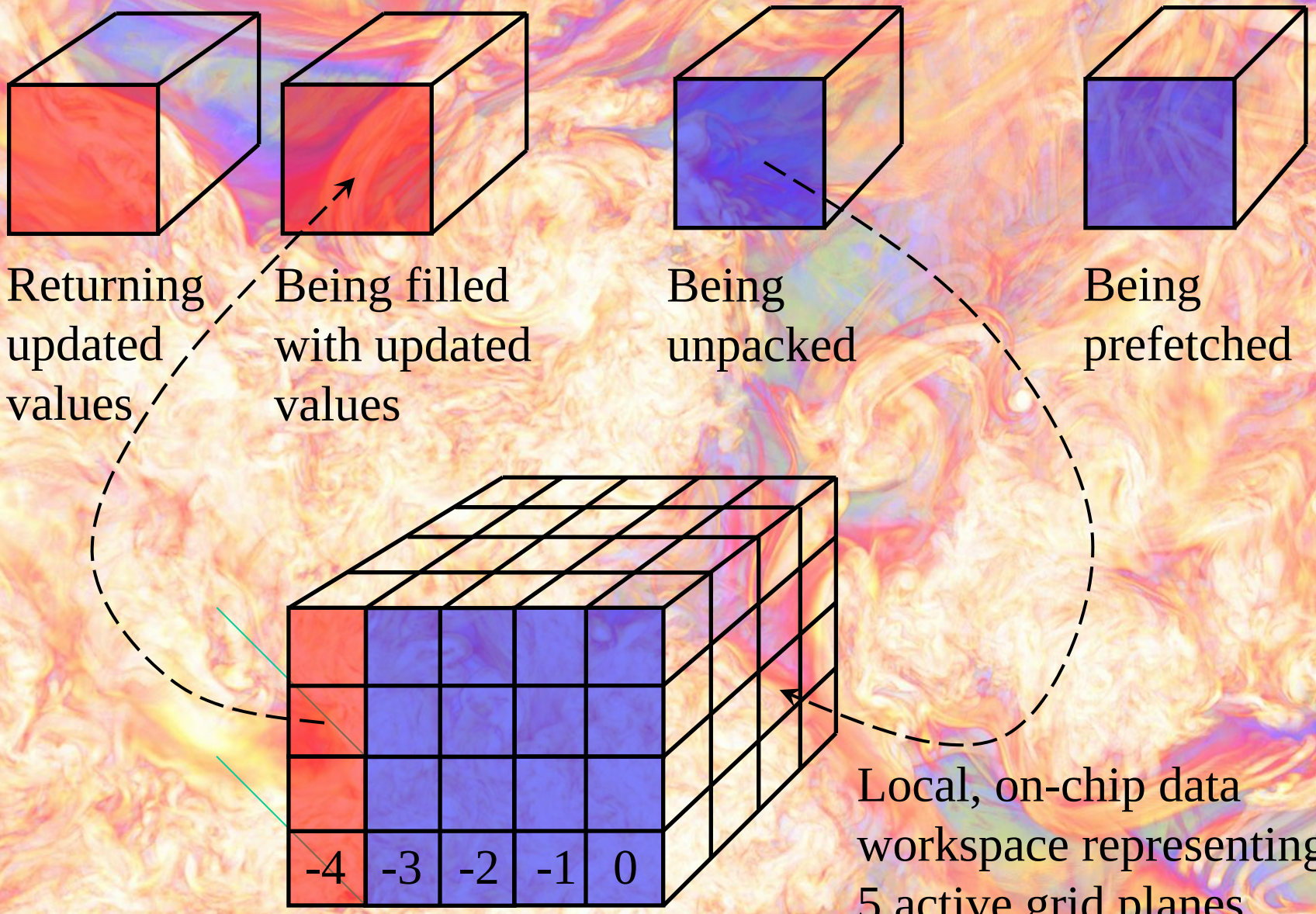
We prefetch one sugar cube while we unpack and operate on the previous one cached in our local store.

We write back one updated sugar cube while we fill in values in another one cached in our local store.

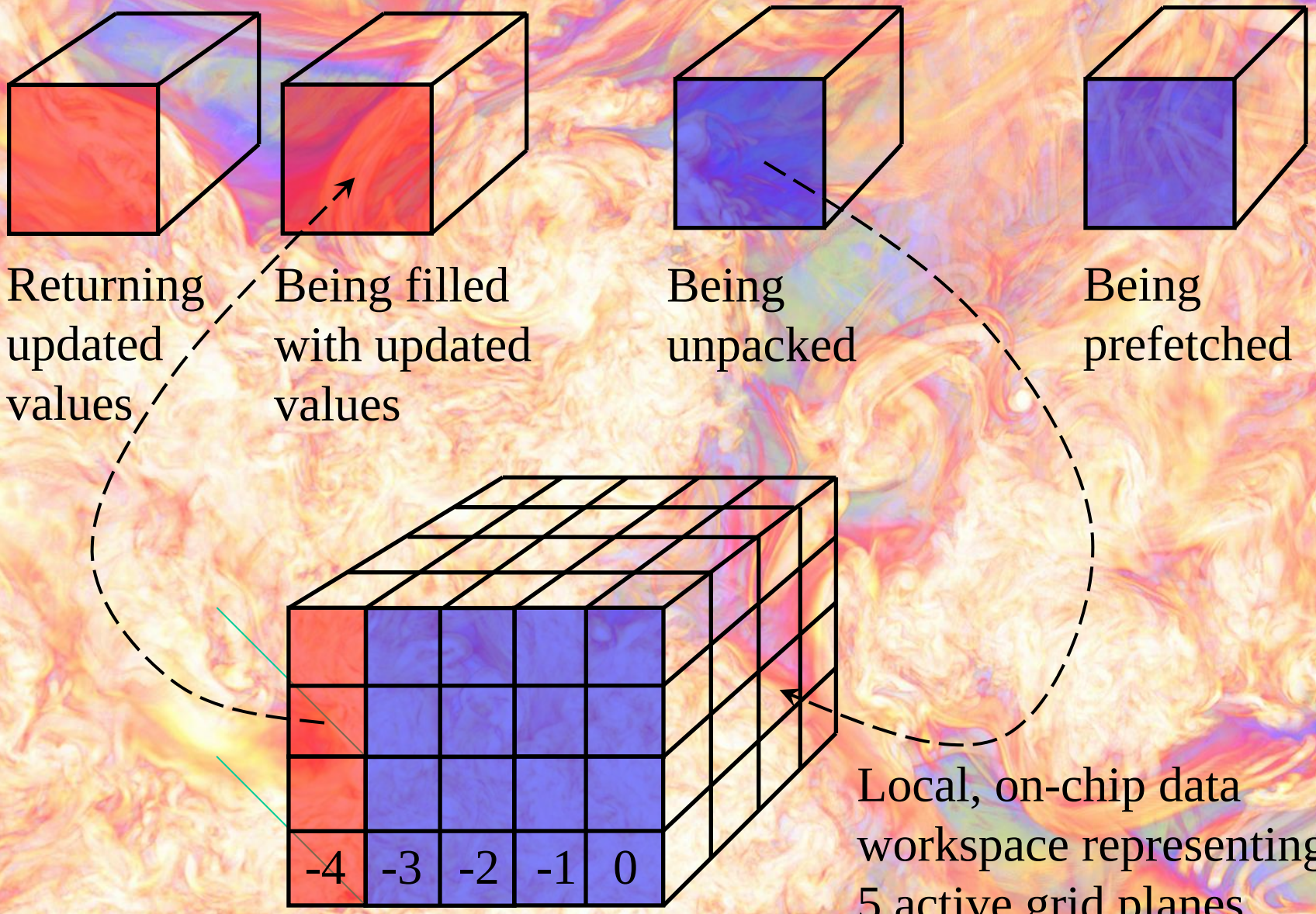


The sugarcube records shown at the bottom are in main memory, while those above are in the SPU local store.

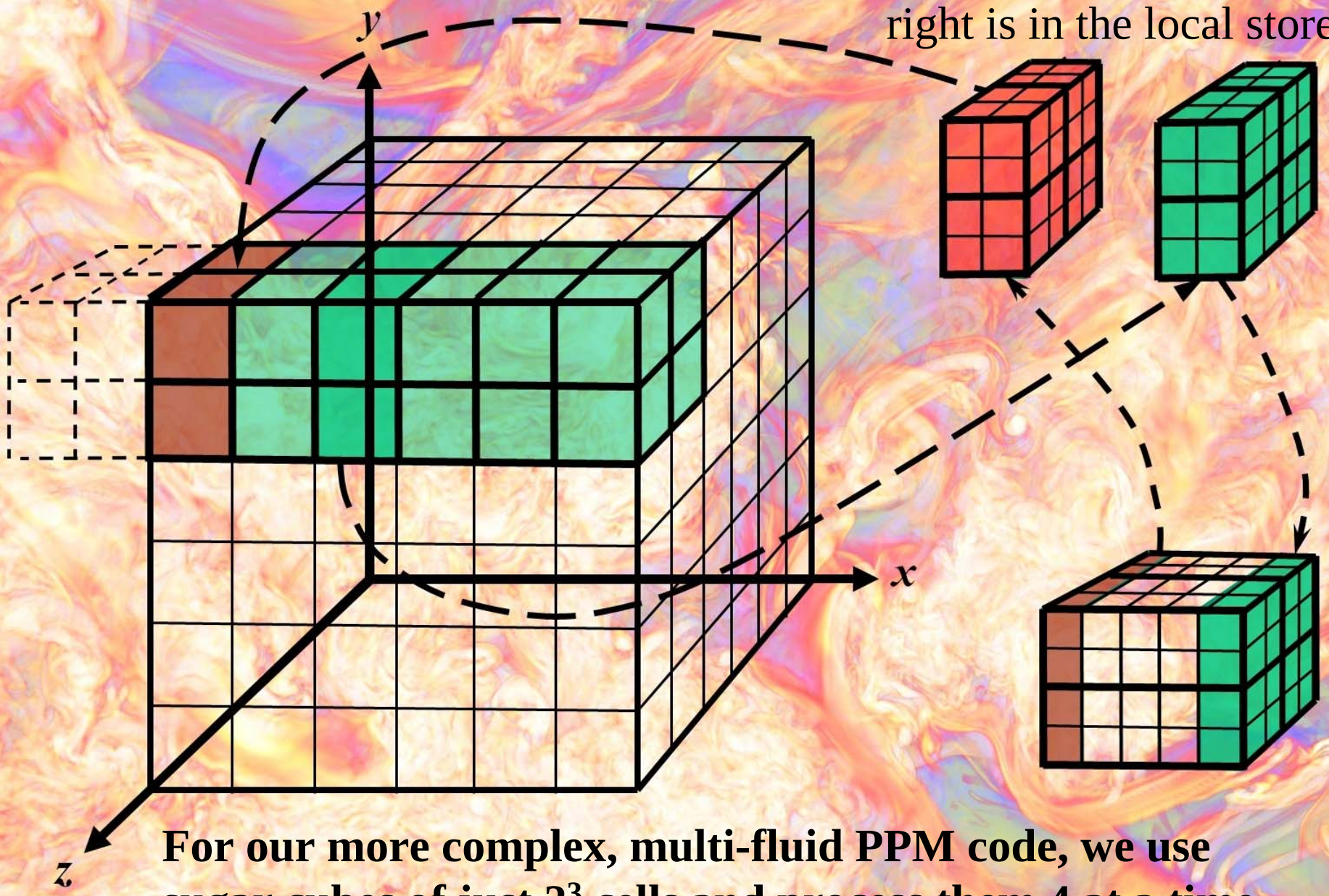
We build a grid-plane processing pipeline in the local store.



But how on earth do we write this program?



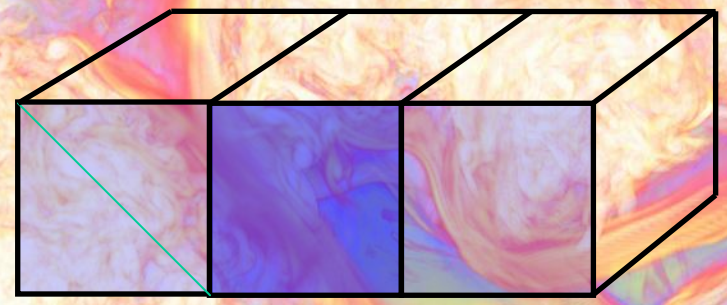
The brick at the left is in main memory, while the data indicated at the right is in the local store.



For our more complex, multi-fluid PPM code, we use sugar cubes of just 2^3 cells and process them 4 at a time.

The strategy I will set out in this tutorial goes as follows:

1. Write a program for updating all the values in just a single grid cube.
2. This will require a ghost cube on each end for the X-pass.
3. Debug the program by making the single grid cube have a large number of cells on each side, such as 32, 64, or 128. This allows something debuggable to happen inside the grid cube.
4. Transform the program via a standard procedure.
5. We are developing a code transformation tool to perform this conversion from “slow Fortran” to “fast Fortran”



Template PPM Code:

For simplicity, we will leave out the MPI (you know how to do that already):

We will look at a code for a single grid brick.

We will use the sugar-cube data structure.

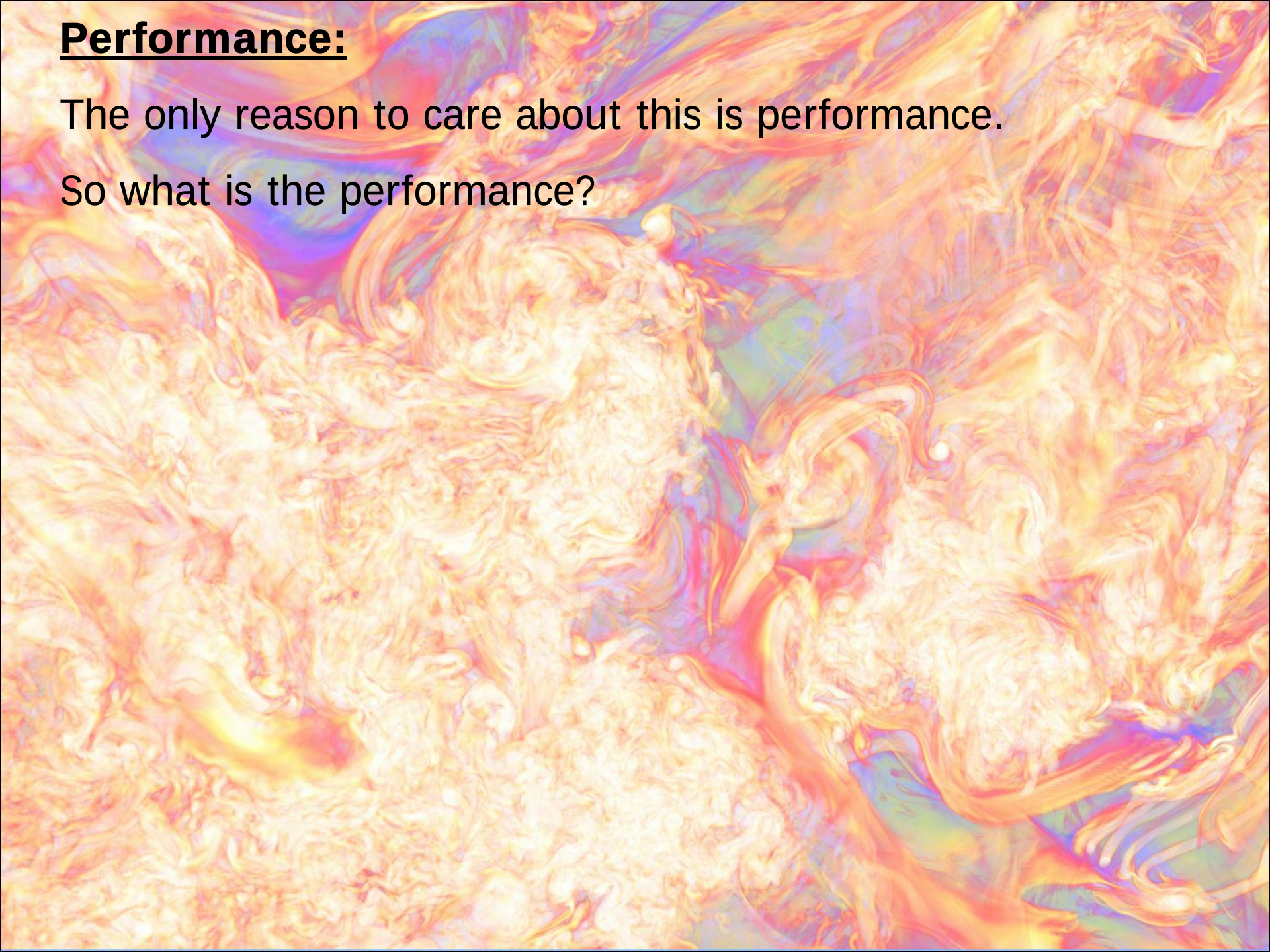
We will implement the parallel update of strips of sugar cubes using OpenMP, so that this code runs everywhere.

The OpenMP implementation is extremely similar to the PPU-SPU implementation for Cell, so this will be the Fortran equivalent of the code for a single Cell processor.

We will focus on how we generate what becomes the highly pipelined code for a single SPU.

Then we will show how we coordinate SPUs with the PPU.

This will take all our time.



Performance:

The only reason to care about this is performance.

So what is the performance?

PPMsloflo Slow & Fast Fortran Performance
10/12/07

PPMsloflo At Nx=128:

Adds/cell = 1065, Mults/cell = 1041, Rsqrts/cell = 3.19

Recips/cell = 43.22, Cvmgms/cell = 575. Variables/cell = 6.

3.0 GHz Clovertown, 4 MB cache, Slow Fortran:

Nx = 8,	6114 Mflop/s;	2990 flops/cell.
Nx = 16,	6078 Mflop/s;	2613 flops/cell.
Nx = 32,	2111 Mflop/s;	2426 flops/cell.
Nx = 64,	1546 Mflop/s;	2350 flops/cell.
Nx = 128,	1328 Mflop/s;	2302 flops/cell.

3.0 GHz Clovertown, 4 MB cache, Fast Fortran:

Nx = 8,	7202 Mflop/s;	2984 flops/cell.
Nx = 16,	7216 Mflop/s;	2603 flops/cell.
Nx = 32,	7201 Mflop/s;	2418 flops/cell.
Nx = 64,	6289 Mflop/s;	2322 flops/cell.
Nx = 128,	6178 Mflop/s;	2281 flops/cell.

The Slow Fortran computation will not fit into a Cell SPU local store at any of the listed sugar cube sizes. On Cell there is no choice.

Comments on PPMsloflo Slow & Fast Fortran Performance

The slow Fortran code is very much easier to write, debug, modify, and maintain.

When the grid brick is as small as 16^3 cells, the entire update fits into the 2 MB cache, and the performance jumps up to 6.1 Gflop/s.

The performance figures on the previous slide are for doing an entire problem of the quoted grid size, not just a subdomain of a larger problem.

When the problem domain is only a cube 16 cells on a side, the entire grid fits into the on-chip cache. This performance is not, however representative of what we will achieve with this code when we ask it to update a sequence of grid cubes of 16^3 cells each which make up a larger problem domain. That task will require a great deal of traffic between the CPU and the main memory. The performance for that much harder and much more useful task is shown on the next slide.

PPMsloflo Slow & Fast Fortran Performance
5/14/08
Nx = size of problem domain, N = size of grid briquette
OMPm indicates OpenMP with m threads

PPMsloflo At Nx=128:
Adds/cell = 1065, Mults/cell = 1041, Rsqrts/cell = 3.19
Recips/cell = 43.22, Cvmgms/cell = 575. Variables/cell = 6.
A few additional flops come from preparation of the output data at intervals of 40 time steps.

3.0 GHz Clovertown, 4 MB cache, Slow Fortran:

Nx = 128, OMP2	N = 16,	3508 Mflop/s/core;	2647 flops/cell.
Nx = 128, OMP4	N = 16,	3478 Mflop/s/core;	2645 flops/cell.
Nx = 128, OMP8	N = 16,	1556 Mflop/s/core;	2644 flops/cell.

3.0 GHz Clovertown, 4 MB cache, Fast Fortran:

Nx = 128, OMP2	N = 4,	6065 Mflop/s/core;	2314 flops/cell.
Nx = 128, OMP4	N = 4,	6104 Mflop/s/core;	2309 flops/cell.
Nx = 128, OMP8	N = 4,	5993 Mflop/s/core;	2309 flops/cell.

Comments on PPMsloflo Slow & Fast Fortran Performance

The slow Fortran code is very much easier to write, debug, modify, and maintain.

When we ask all 8 CPU cores in the 2 processors sharing a common memory in a PC workstation to cooperatively update a suefully large problem (128^3 cells is actually not large at all), then we find the true limitation of Slow Fortran – the main memory bandwidth that it unreasonably requires.

In this case we discover that for Slow Fortran, the 8 cores solve the fluid flow problem more slowly than if we leave 4 of the 8 cores completely idle.

However, for Fast Fortran, each of the 8 cores delivers its full potential in this cooperative computation, so that all 8 taken together outperform the same 8 cores running the same problem in Slow Fortran by a dramatic factor of 4.41 (accounting both for the higher Mflop/s rate of each core and also the smaller number of flops that need to be performed in the Fast Fortran implementation).

PPMsloflo Slow & Fast Fortran Performance
2/1/08 Nx = size of entire problem domain

PPMsloflo At Nx=128:

Adds/cell = 1065, Mults/cell = 1041, Rsqrts/cell = 3.19
Recips/cell = 43.22, Cvmgms/cell = 575. Variables/cell = 6.

3.0 GHz Clovertown, 4 MB cache, Slow Fortran (only 1 thread):

Nx = 8, 6114 Mflop/s; 2990 flops/cell.
Nx = 16, 6078 Mflop/s; 2613 flops/cell.
Nx = 32, 2111 Mflop/s; 2426 flops/cell.
Nx = 64, 1546 Mflop/s; 2350 flops/cell.
Nx = 128, 1328 Mflop/s; 2302 flops/cell.

16 Slow Fortran threads do not achieve this same high performance per core.

3.2 GHz Cell SPU, 256 KB cache, Fast Fortran (16 SPUs coop):

Nx = 32, 7.73 Gflop/s; 2418 flops/cell.
Nx = 64, 5.50 Gflop/s; 2322 flops/cell.
Nx = 128, 5.68 Gflop/s; 2281 flops/cell.

16 threads all beat on the shared memory here.

We believe the performance jumps up for the small bricks on Cell because there are suddenly no TLB misses.

PPMsloflo Slow Fortran Laptop Performance 11/4/07

PPMsloflo At Nx=128:

Adds/cell = 1065, Mults/cell = 1041, Rsqrts/cell = 3.19

Recips/cell = 43.22, Cvmgms/cell = 575. Variables/cell = 6.

2.4 GHz Core Duo, 4 MB cache, Slow Fortran:

Nx = 128, N = 8, No OMP	4154 Mflop/s; 3021 flops/cell.
Nx = 128, N = 16, No OMP	3862 Mflop/s; 2643 flops/cell.
Nx = 128, N = 16, OMP2	5066 Mflop/s; 2643 flops/cell.
Nx = 128, N = 16, OMP1	3336 Mflop/s; 2643 flops/cell.
Nx = 128, N = 128, No OMP	1328 Mflop/s; 2302 flops/cell.

I spent a week trying every possible way to improve performance on this slow Fortran expression by using both cores in the Laptop CPU. The key turned out to be putting the working data for each thread onto the stack rather than in a “threadprivate” common block. This is the difference between “OMP1” and “OMP2” above.

The 5 Gflop/s looks pretty good for a laptop, but . . .

PPMsloflo Fast Fortran Laptop Performance
11/4/07

PPMsloflo At Nx=128:
Adds/cell = 1065, Mults/cell = 1041, Rsqrts/cell = 3.19
Recips/cell = 43.22, Cvmgms/cell = 575. Variables/cell = 6.

Stack = private workspace on stack.

2.4 GHz Core Duo, 4 MB cache, Fast Fortran:

Nx = 64, N = 4, OMP2, stack 9754 Mflop/s; 2354 flops/cell.
Nx = 64, N = 4, No OMP, stack 5024 Mflop/s; 2354 flops/cell.
Nx = 128, N = 4, OMP2, stack 9539 Mflop/s; 2308 flops/cell.
Nx = 256, N = 4, OMP2, stack 9037 Mflop/s; 2270 flops/cell.

Who would have believed that my laptop could do 9.75 Gflop/s?

Here we see the benefit of Fast Fortran for a multicore Intel CPU.
It reduces the main memory bandwidth requirement.

PPMsloflo Fast Fortran Workstation Performance 11/4/07

PPMsloflo At Nx=128:

Adds/cell = 1065, Mults/cell = 1041, Rsqrts/cell = 3.19

Recips/cell = 43.22, Cvmgms/cell = 575. Variables/cell = 6.

Stack = private workspace on stack.

The per-core performance for 8 threads is only 3.5% lower than for 4 threads. Hence there can be very little contention for the main memory bus in this parallel application.

3.0 GHz Clovertown, 4 MB cache, Fast Fortran:

Nx = 128, N = 4, OMP4, stack 23.89 Gflop/s; 2292 flops/cell.

Nx = 128, N = 4, OMP8, stack 46.09 Gflop/s; 2292 flops/cell.

Nx = 256, N = 4, OMP8, stack 46.31 Gflop/s; 2270 flops/cell.

Nx = 512, N = 4, OMP8, stack 42.00 Gflop/s; ???? flops/cell.

What you get on four 8-core PC workstations in 4 days of running the 2-fluid PPM on a 512^3 grid OpenMP+MPI:

The following slides show results (at dump 400, 5147 sec.) of deep convection (9 Mm to 30 Mm) in the white-dwarf-like core of a 2 solar mass star near the end of its life.

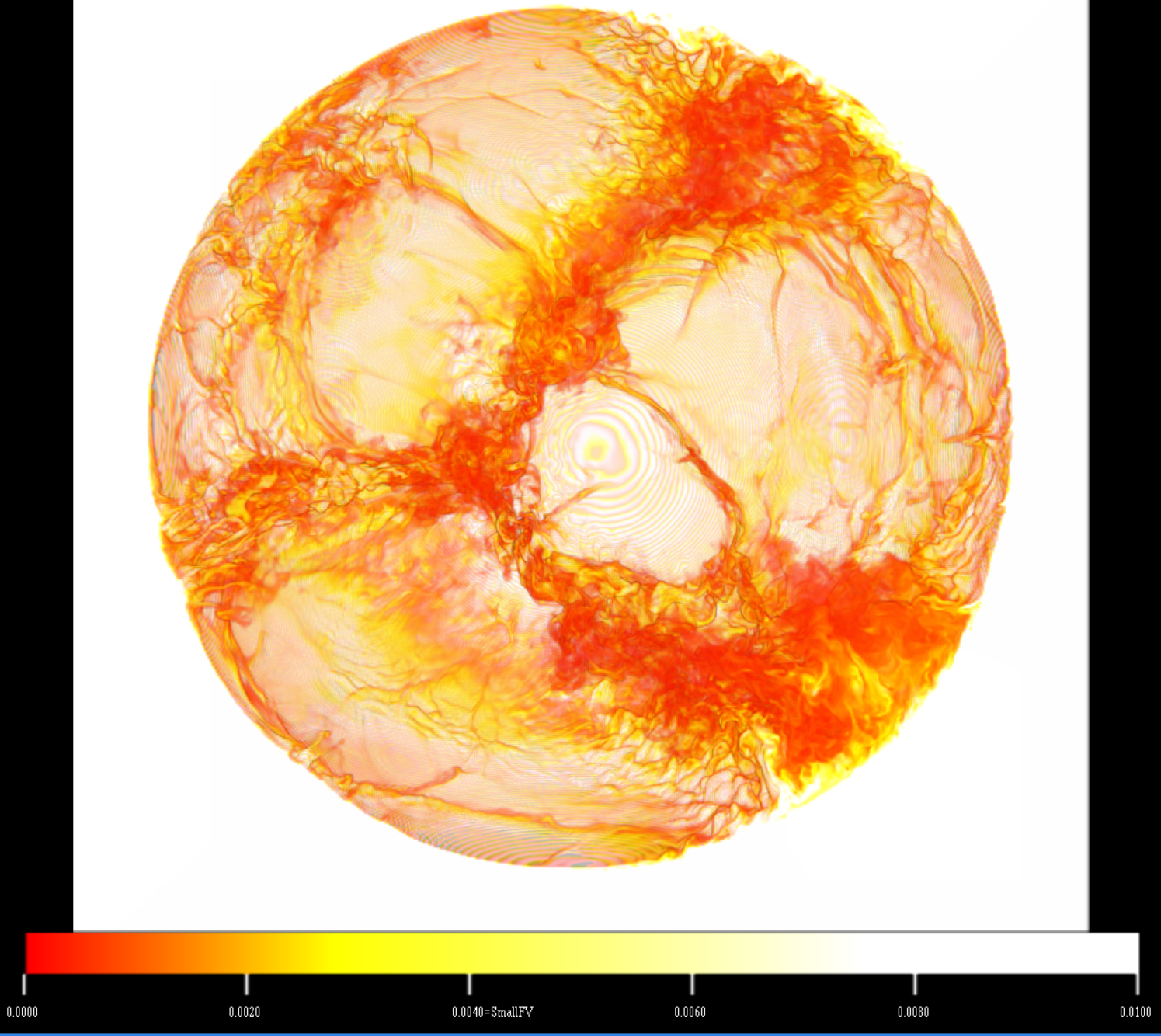
The convection zone above the helium burning shell has engorged all the previously processed material, so that it now begins to entrain the 2.6 times more buoyant unburned hydrogen fuel.

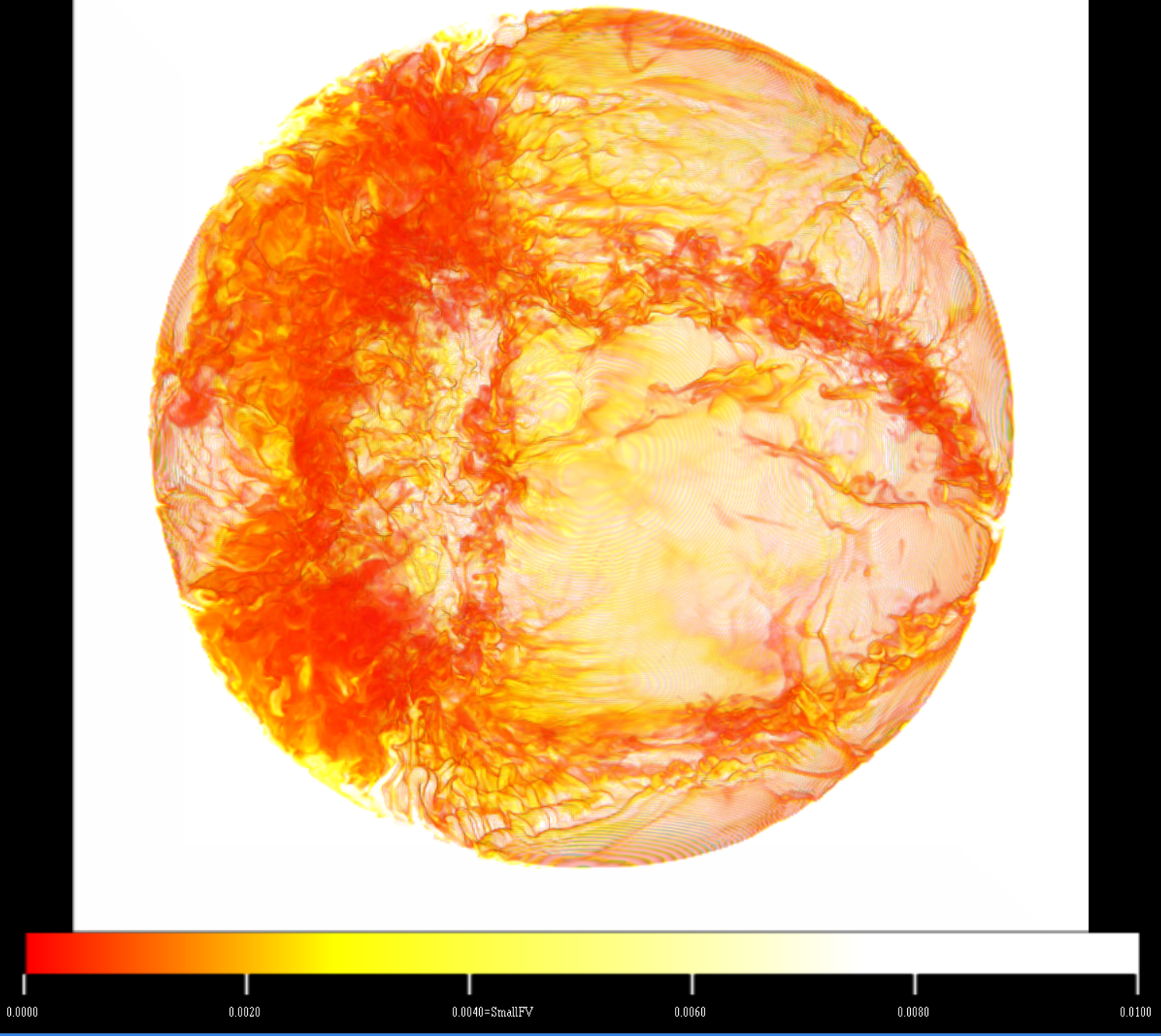
The mixing fraction of unburned hydrogen is shown in two opposite hemispheres on the next 2 slides, and a convection pattern that is nearly global in scale is revealed.

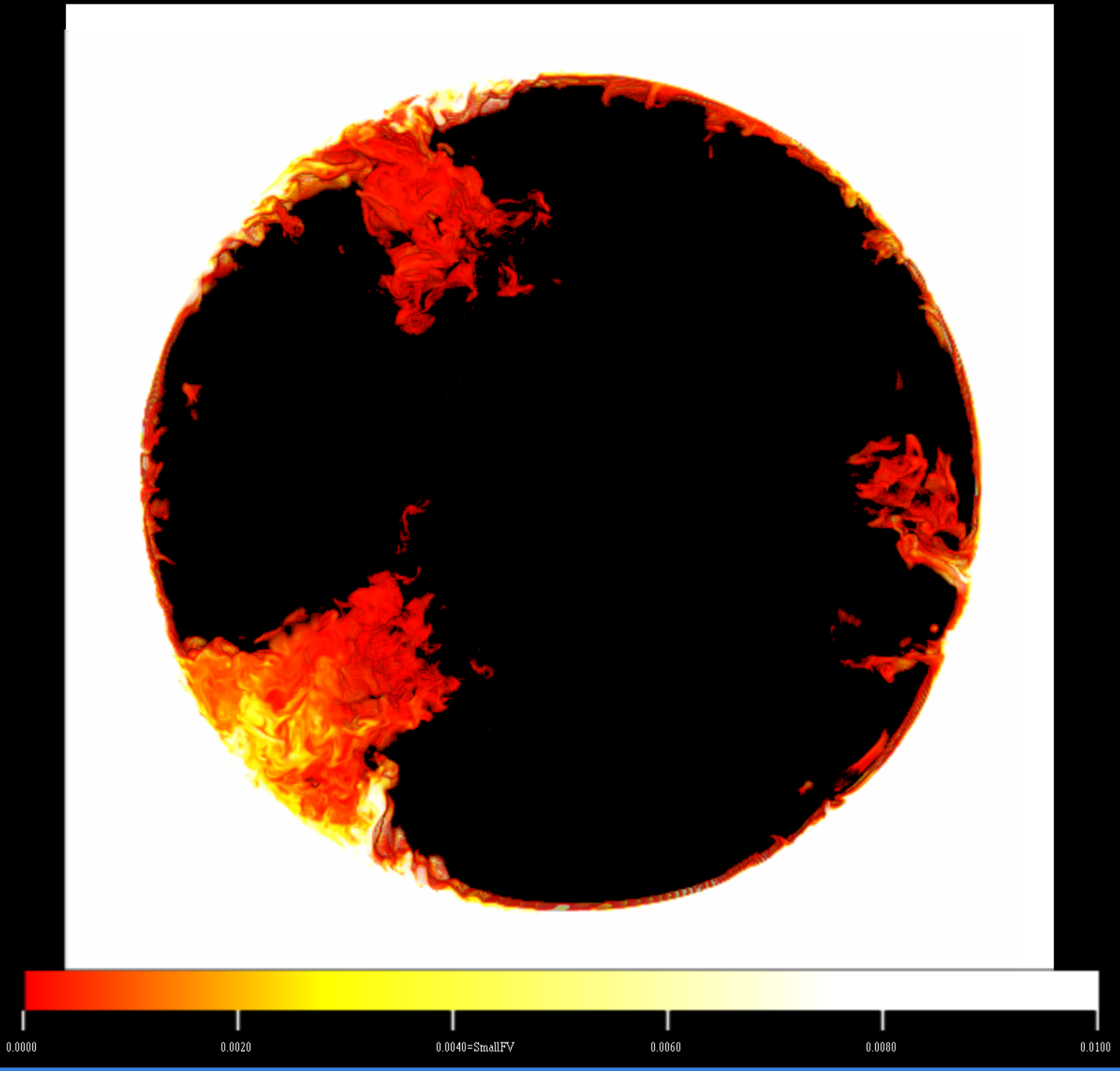
Darker colors show the bottoms of descending sheets of cooler gas that separate the 4 very large convection cells.

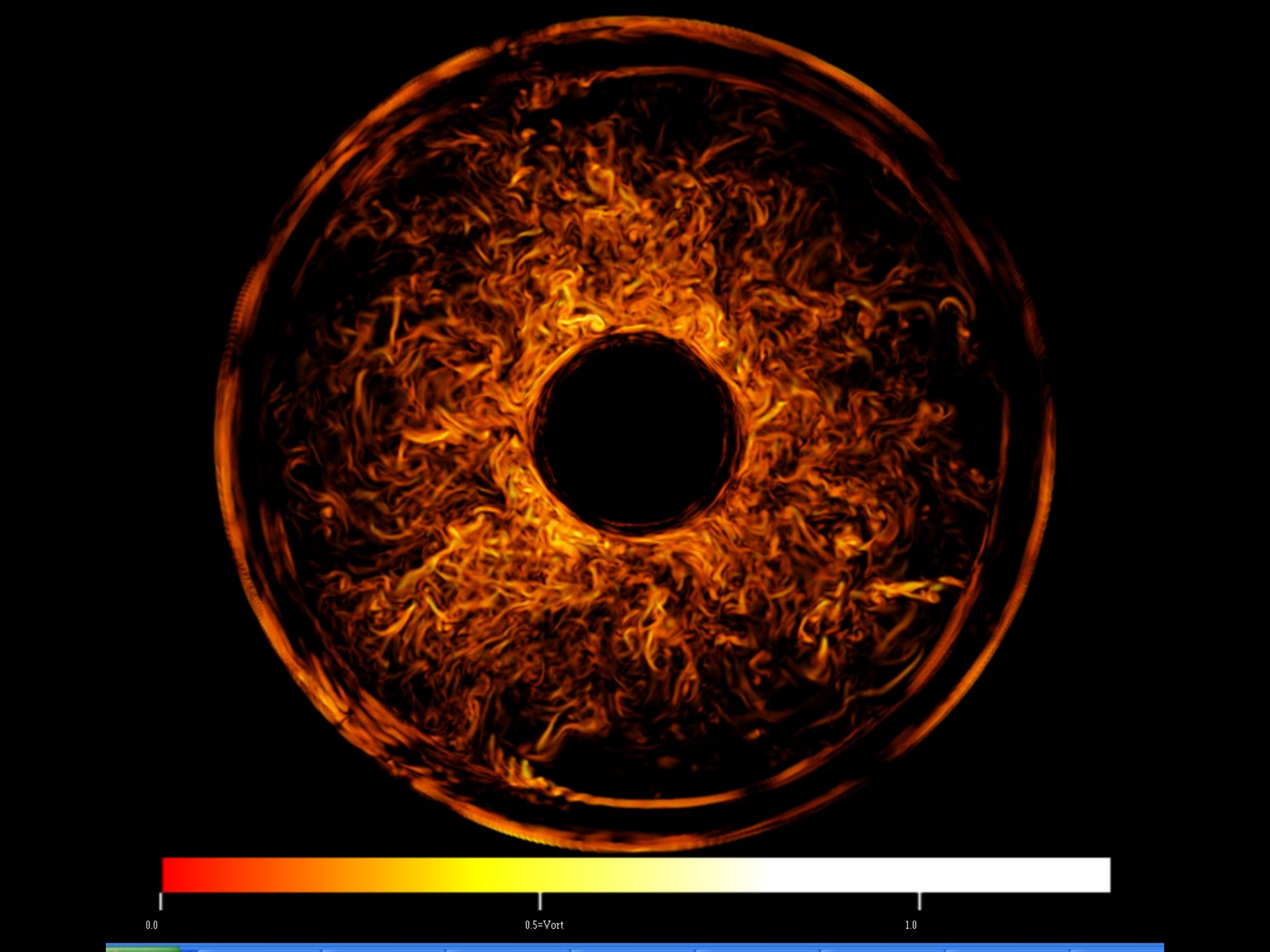
A thin slice through the volume shows the depth to which the entrained lighter gas descends – well beyond the level where vigorous nuclear burning would take place.

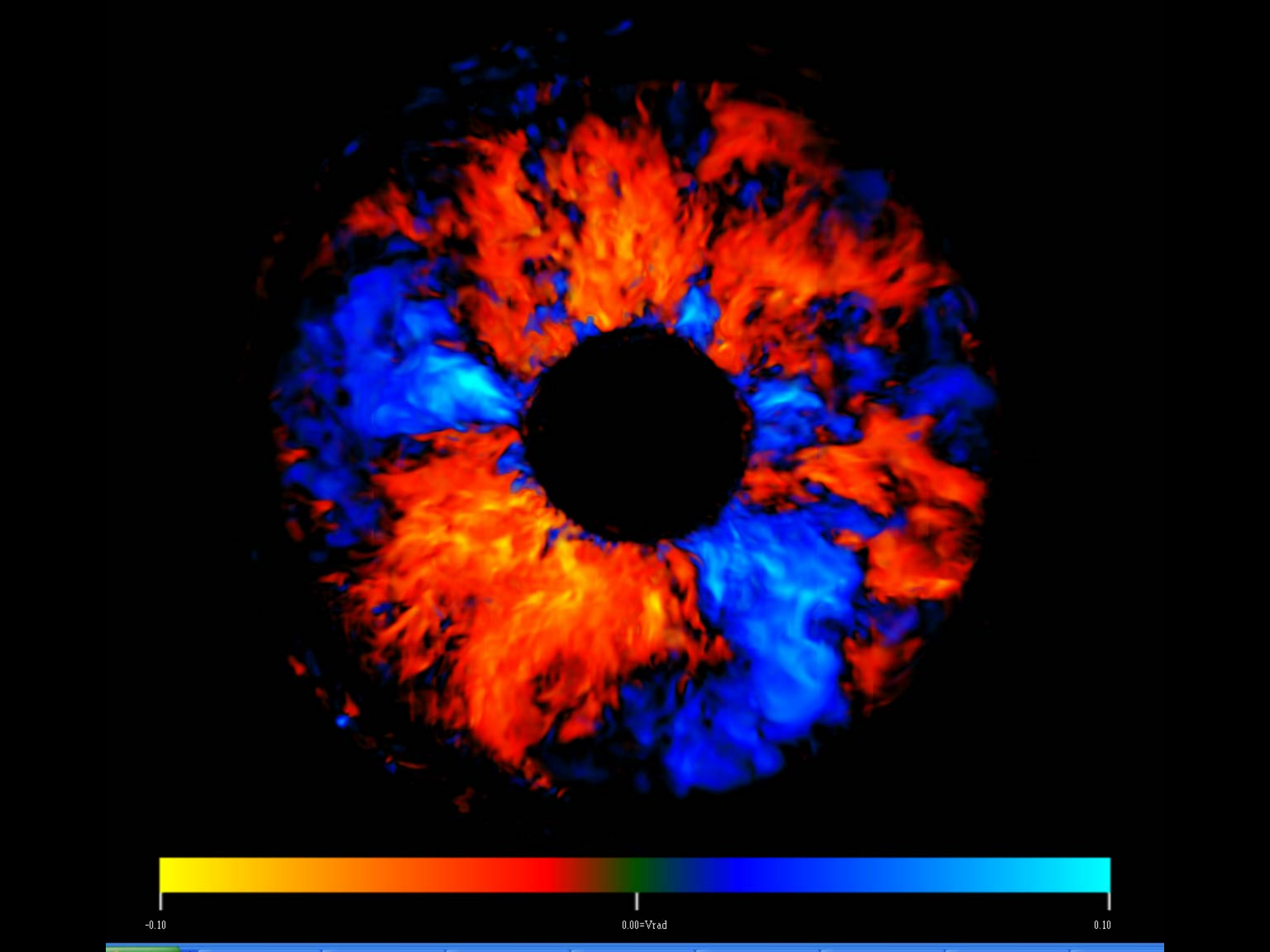
Finally, the magnitude of vorticity and the radial component of the velocity are shown in this same slice.


0.0000
0.0020
0.0040=SmallFV
0.0060
0.0080
0.0100


0.0000
0.0020
0.0040=SmallFV
0.0060
0.0080
0.0100


0.0000
0.0020
0.0040=SmallFV
0.0060
0.0080
0.0100


0.0
0.5=Vort
1.0


-0.10
0.00=Vrad
0.10

32-bit arithmetic is just fine:

A key to high performance on all of today's microprocessor CPUs is 32-bit arithmetic.

Codes must be carefully written to make this practical.

On the following slides, we show comparisons of the same single-mode Rayleigh-Taylor test problem (IWPCTM-11 test problem 1) computed first with 64-bit arithmetic and then in the second of each slide pair with 32-bit arithmetic.

The heavy gas was 5/3 as dense as the light one in this problem, and we view the developing flow along a diagonal at times noted in each slide. Here the time unit is the sound crossing time of the width of the problem domain in the lighter fluid at the midplane in the initial state.

The grid used in each case was $64 \times 64 \times 512$, and a volume rendering of the mixing fraction of the two fluids is shown.

32-bit arithmetic is just fine:

This flow of the gas of density 5/3 initially superposed above a gas of density 1 in a gravitational field is highly unstable.

The physical instability will exponentially amplify tiny differences in the fluid states of the two simulations as time progresses.

Nevertheless, blinking the images back and forth reveals essentially no differences until 37.5 sound crossing times, when the instability has progressed far into the nonlinear regime.

Differences between the two simulations are not really noticeable until 62.5 sound crossing times, and even at this time they are only minor differences of detail.

We conclude that 32-bit arithmetic is just fine for this problem with this very carefully written code that makes this OK.

t = 0

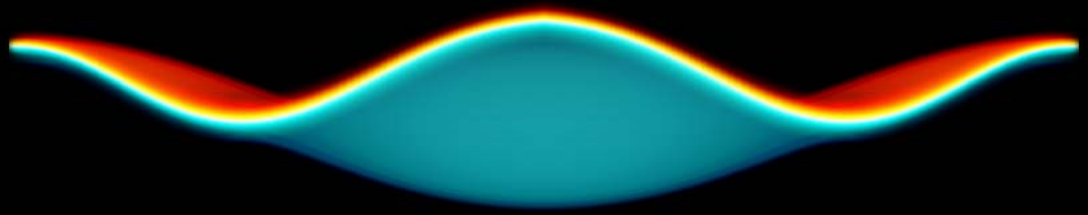
64-bit

t = 0

32-bit

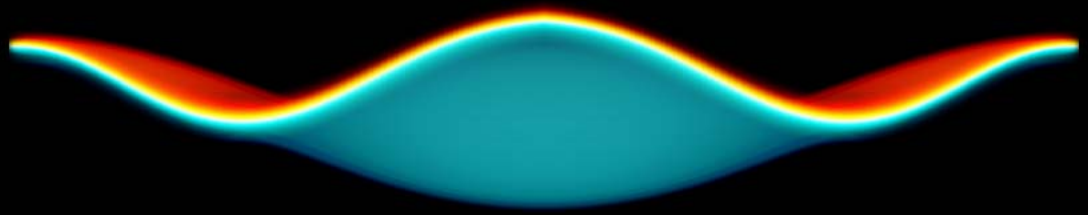
t = 12.5

64-bit



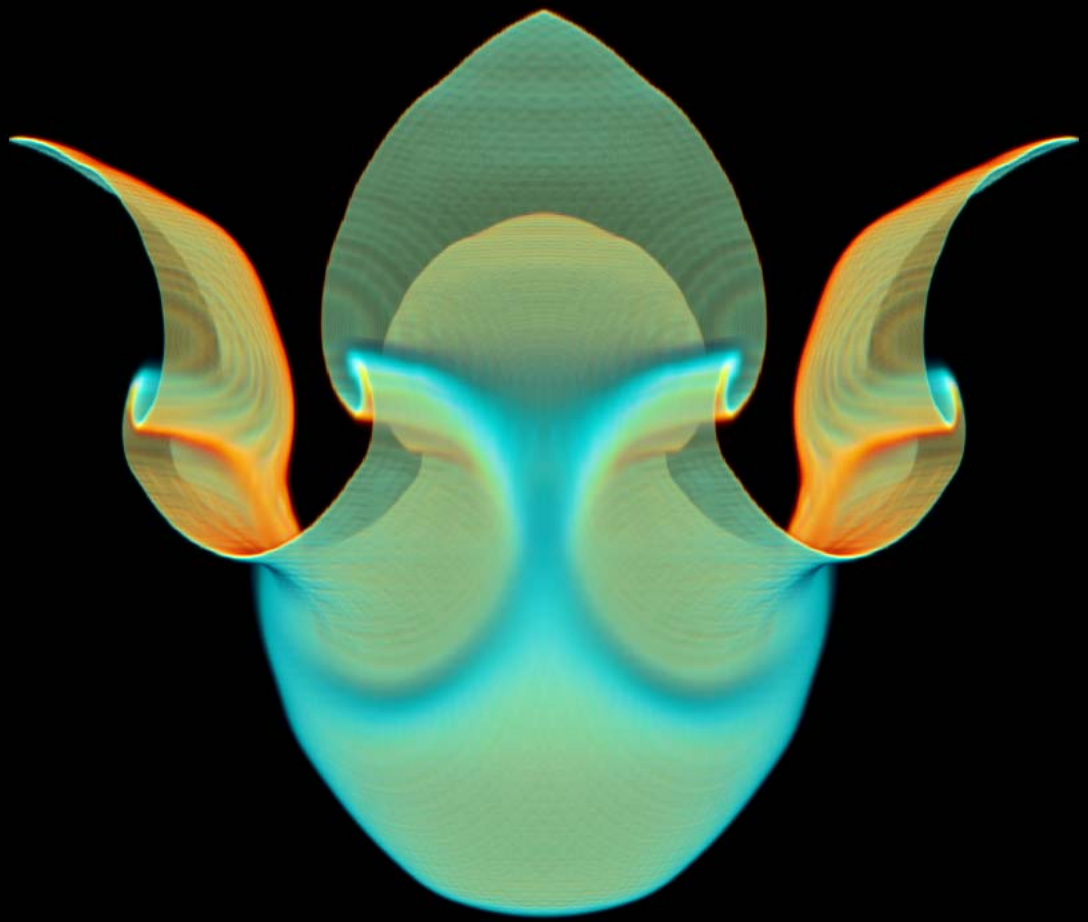
t = 12.5

32-bit



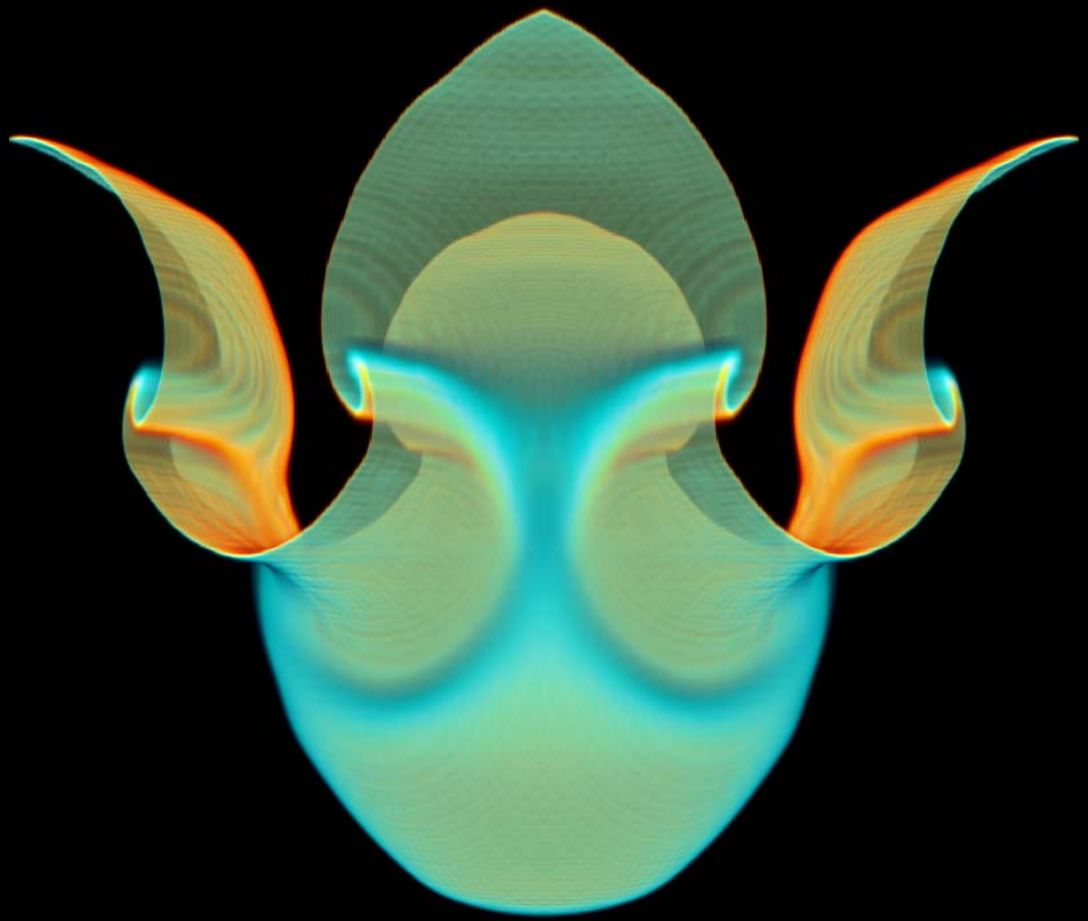
t = 25

64-bit



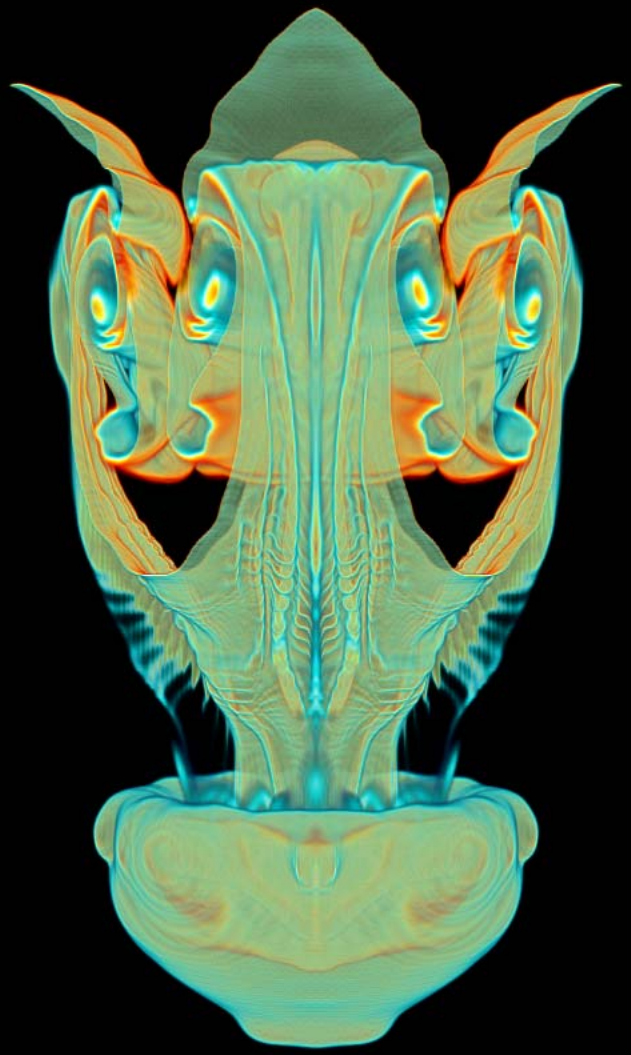
t = 25

32-bit



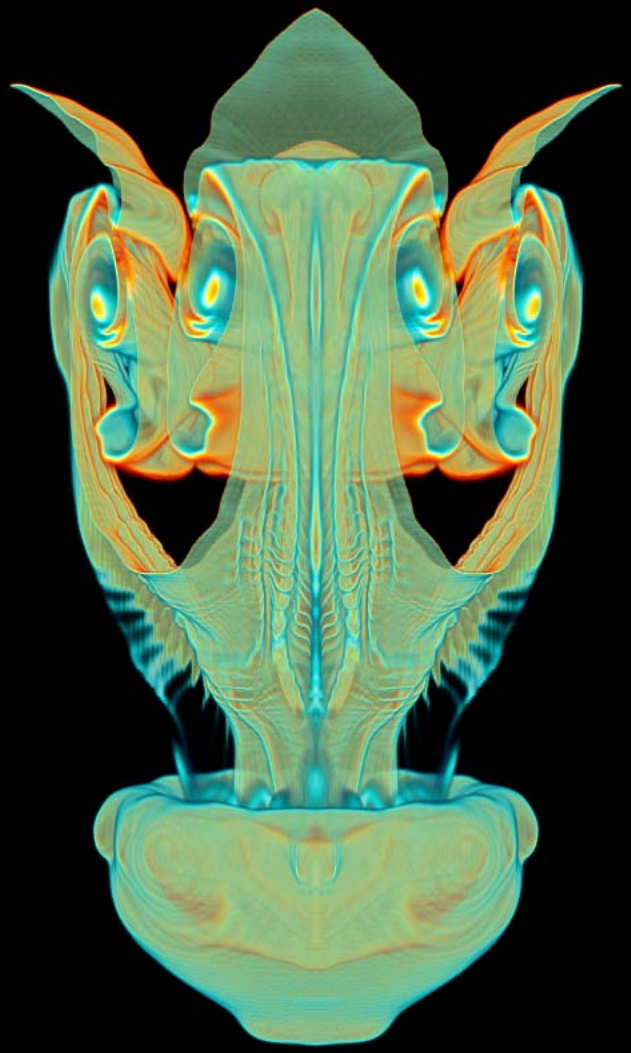
t = 37.5

64-bit



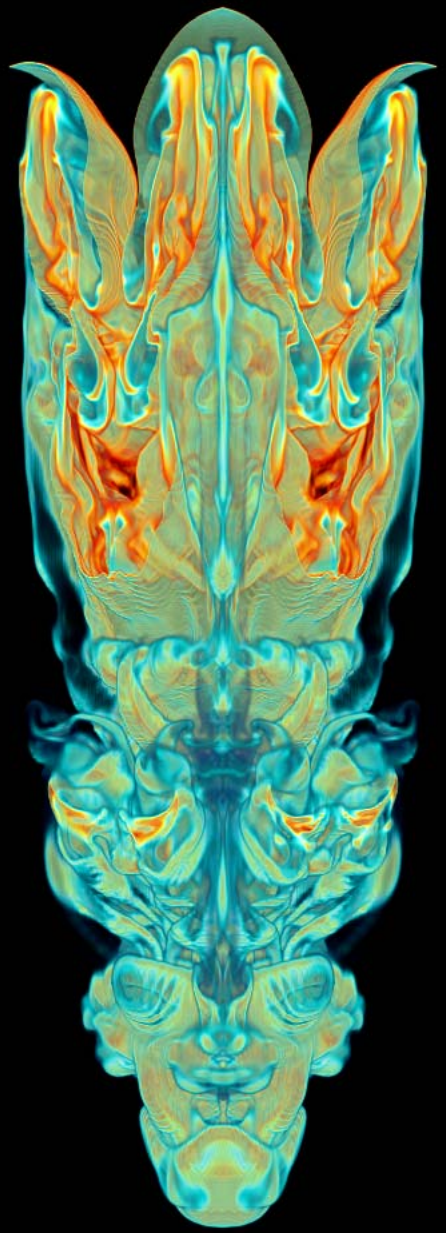
t = 37.5

32-bit



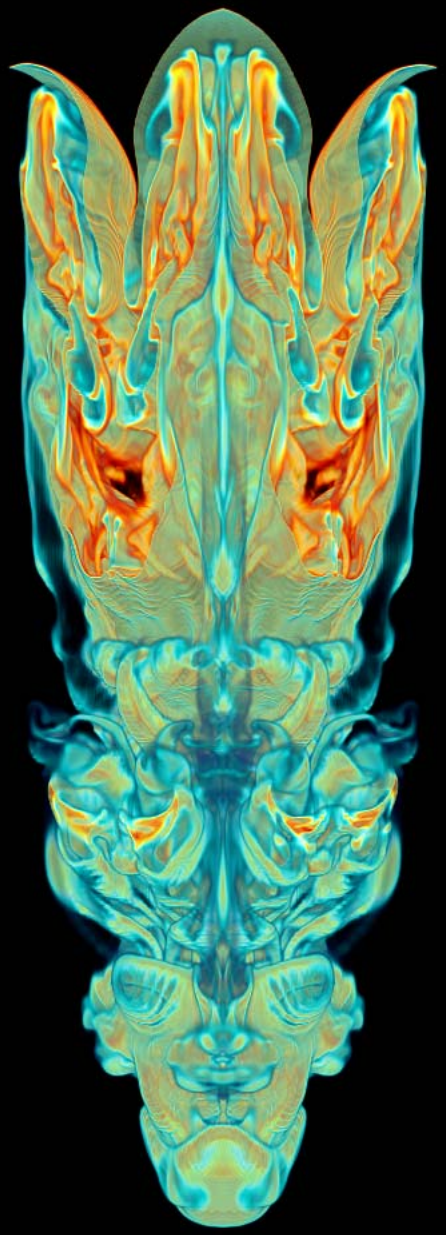
t = 50

64-bit



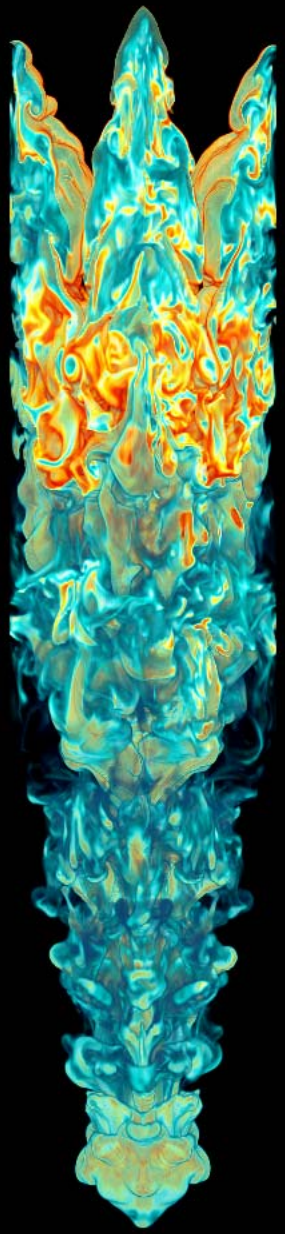
t = 50

32-bit



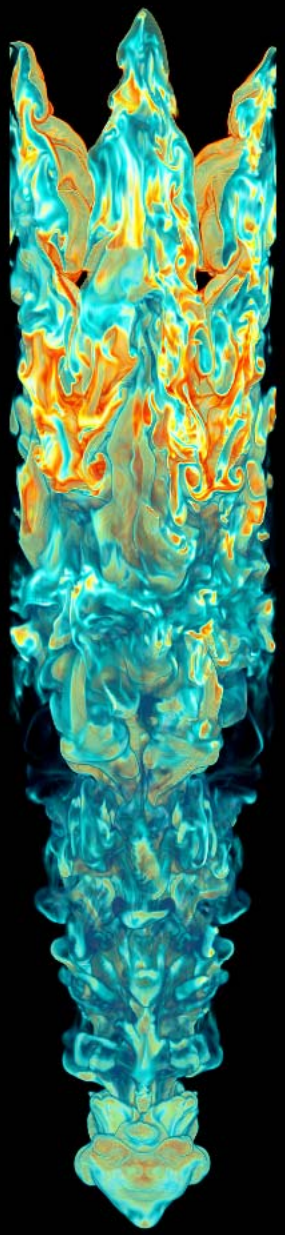
t = 62.5

64-bit



t = 62.5

32-bit



Slow Fortran to Fast Fortran Concept:

Write the program in the simplest possible way, regardless of how inefficiently it will execute when simply compiled with a standard compiler and run on a single processor.

Compute all intermediate quantities on the full 3-D grid, regardless of the waste and inefficiency this implies.

Debug the code for correctness exploiting this simple form.

When the code is correct, ***transform it automatically*** from this "Slow Fortran" expression ***into "Fast Fortran."***

Compile the Fast Fortran to run on mainstream processors as a single-processor module in a parallel program.

Transform the Fast Fortran into Cell-C, with automatic translator, and hand it to the GCC compiler for the Cell SPU.

Use this SPU-module in a parallel Cell program.

Operational Details:

By changing parameters in dimension statements, the Slow Fortran program can update the entire grid.

On modern laptops this will allow the grid to be as large as ***128^3 cells***, which is more than sufficient ***for debugging***.

Performance will be in the range from ***640 to 1330 Mflop/s***, so that test runs do not take too long.

For tiny domains, such as 16^3 cells, a mainstream processor cache will contain the whole data context, and performance can jump as high as 4 Gflop/s.

An outer program of the same type that supports parallel execution of the Fast Fortran module can also support parallel execution of the Slow Fortran one on bricks of 16^3.

OpenMP can be introduced as well, with threads pinned to cores, as long as temporary arrays are on the stack.

Walk through outer code:

Although this was planned, we did not do this at the March 12 tutorial.

The outer code has the MPI message passing, and is not a programming style that is unfamiliar to this community.

Therefore we skipped this planned section.

Walk through Xpass1 code:

This exercise also was skipped, as this is just a sequence of calls to vanilla Fortran subroutines that is also not unusual.

Walk through PPMsloflo code:

This exercise we did actually do. This is the single subroutine that includes essentially all the computation of the code.

Our present code uses the Fortran pre-processor to produce from a single source three separate codes, depending upon the settings of various flags:

1. The code for the Cell processor SPU, which consists essentially entirely of the single routine PPMsloflo (a bit over 5000 Fortran lines, not counting comments).

2. The code for the PPU, which coordinates the SPUs and signals the Opteron that MPI messages have been fully composed in their buffers in the blade memories. The PPU also computes the initial state for the simulation.

3. The Opteron code, which does all MPI library calls and handles all I/O, including output and restart dumps.

Slow Fortran Version of PPMsloflo:

3-D Loop Nests:

Code “basic block” is a 3-D loop nest of the form below, in which ny*nz is an integer multiple of 4.

Scalar temporaries are used extensively in such loops.

```
      do i = 1+ioffl-nbdy,nx+nbdy-ioffr
      do k = 1,nz
!DEC$ VECTOR ALWAYS
!DEC$ VECTOR ALIGNED
      do j = 1,ny
      al(j,k,i) = (a(j,k,i-1) + a(j,k,i)) * .5
      enddo
      enddo
      enddo
```

Slow Fortran 2:

Order of 3-D Loop Nests:

Loop nests are ordered so that the sum ioffl+ioffr increases from one loop nest to the next. This is a completely natural ordering for almost all numerical algorithms, as it expresses an informational “light cone” arising from causality.

```
      do i = 1+ioffl-nbdy,nx+nbdy-ioffr
      do k = 1,nz
!DEC$ VECTOR ALWAYS
!DEC$ VECTOR ALIGNED
      do j = 1,ny
      al(j,k,i) = (a(j,k,i-1) + a(j,k,i)) * .5
      enddo
      enddo
      enddo
```

Slow Fortran 3:

Syntax Restrictions (to ease automatic translation):

Continuations allowed only in SUBROUTINE statements.

Only 2 permitted forms of vectorizable logic, equivalent to Cray's original cvmgm and cvmgz intrinsics.

No more than 2 arithmetical operators per line.

Intrinsic function call requires separate line.

Arithmetic must read correctly left to right with parentheses ignored.

No common blocks allowed in code to be translated.

3-D arrays of a standard, conformal shape may be designated within storage arrays by means of equivalences.

Restrictions apply only to code to be automatically translated to Cell-specific C for the SPU.

Slow Fortran 4:

Array Dimensioning Using Parameters, Not Variables:

dimension A (ny, nz, 1-nbdy:nx+nbdy)

If A is a subroutine argument, it is good programming practice for nx, ny, and nz to be arguments also.

However, for performance of the compiled code, it is best for nx, ny, and nz to be parameters set to constant values.

All arrays A that are locally allocated must be allocated on the stack, if an OpenMP thread executing this code is to perform

This may require a compile-time specification of the stack size limit for the linker.

These rules of Slow Fortran guarantee proper data alignment, generated by the mainstream Fortran compiler, and they avoid “known” previous bugs in the Intel compiler.

Slow Fortran 5:

Syntax Restrictions, continued:

Each serial data copy from or to main memory is expressed in a separate loop, preceded by a DMA directive and with this expression made possible by an equivalence statement.

Example:

```
      dimension   Dvar(ny,nz,1-nbdy:nx+nbdy,nvars)
      dimension   Dvars(ny*nz*(nx+2*nbdy)*nvars)
      equivalence (Dvars,Dvar)
      lenvars = ny*nz*(nx+2*nbdy)*nvars
cPPM$ DMA
      do i = 1,lenvars
      Dvars(i) = whatever(ioff+i)
      enddo
cPPM$ END DMA
```

Here Dvars is on the stack and hence will be cache resident.

First PPM Loop Nest:

```
      do i = 1-nbdy,n+nbdy
      do k = 1,n
!DEC$ VECTOR ALWAYS
!DEC$ VECTOR ALIGNED
      do j = 1,n
      pv = p(j,k,i) / rho(j,k,i)
      ceulsq = gamma * pv
      ceul2i(j,k,i) = 1. / ceulsq
      ceul = 1. / sqrt(ceul2i(j,k,i))
      c(j,k,i) = ceul * rho(j,k,i)
      . . .
      enddo
      enddo
      enddo
```

Here we get the sound speeds from the density and pressure.

This is done in all the cells and all the ghost cells.

Second PPM Loop Nest:

```
      do i = 2-nbdy,n+nbdy
      do k = 1,n
!DEC$ VECTOR ALWAYS
!DEC$ VECTOR ALIGNED
      do j = 1,n
      cl(j,k,i) = (c(j,k,i-1) + c(j,k,i)) * .5
      clinv(j,k,i) = 1. / c(j,k,i)
      temp = (p(j,k,i) – p(j,k,i-1)) * clinv(j,k,i)
      drplsl(j,k,i) = (ux(j,k,i) – ux(j,k,i-1)) + temp
      . . .
      enddo
      enddo
      enddo
```

Here we compute Riemann invariant differences. We require 2 planes of results from the previous loop nest. Hence we must execute the inner loops of that nest twice before we may execute the inner loops of this nest for the first time.

Third PPM Loop Nest:

```
      do i = 2-nbdy,n+nbdy-1
      do k = 1,n
!DEC$ VECTOR ALWAYS
!DEC$ VECTOR ALIGNED
      do j = 1,n
      dasppm(j,k,i) = (drplsl(j,k,i) + drplsl(j,k,i+1))
     &                * .5
      a6sppm(j,k,i) = (drplsl(j,k,i) - drplsl(j,k,i+1))
     &                * .5
      . . .
      enddo
      enddo
      enddo
```

Here we compute interpolation coefficients. We require 2 planes of results from the previous loop nest. Hence we must execute the inner loops of that nest twice before we may execute the inner loops of this nest for the first time.

Third PPM Loop Nest, Right Justified:

```
      do i = 3-nbdy,n+nbdy
      do k = 1,n
!DEC$ VECTOR ALWAYS
!DEC$ VECTOR ALIGNED
      do j = 1,n
      dasppm(j,k,i-1) = (drplsl(j,k,i-1) + drplsl(j,k,i))
     &                  * .5
      a6sppm(j,k,i-1) = (drplsl(j,k,i-1) - drplsl(j,k,i))
     &                  * .5
      . . .
      enddo
      enddo
      enddo
```

When we fuse all the outer loops on i we may now just place before the inner loops of this nest a test:

if (i .lt. 3-nbdy) go to 9000

Elimination of Unnecessary Storage:

```
      isave = i2m2
      i2m2 = i2m1
      i2m1 = 12m0
      i2m0 = isave
      if (i .lt. 3-nbdy)   go to 9000
!DEC$ VECTOR ALWAYS
!DEC$ VECTOR ALIGNED
      do jk = 1,n*n
      dasppm(jk,i2m1) = (drplsl(jk,i2m1)
     &                  + drplsl(jk,i2m0)) * .5
      a6sppm(jk,i2m1) = (drplsl(jk,i2m1)
     &                  - drplsl(jk,i2m0)) * .5
      . . .
      enddo
```

Only 3 grid planes of drplsl are ever referenced. Hence all but these are unnecessary. We use integer variables to represent these grid planes, and they act like pointers. Note the barrel shift operation on each outer loop traversal.

Code Transformation Steps:

1. Write code for a single grid briquette.
2. Dimension all variables over the entire briquette + ghosts.
3. Inline all subroutines.
4. Order the loops to reflect causality. Largest extents in index i come first.
5. Right-justify all outermost loops and fuse inner loop pairs.
6. Fuse all outermost loops, inserting jumps to end (9000).
7. Identify live planes for all temporary arrays. Then change references to i-3, etc., to i5m3, or whatever. Revise dimension statements to collapse storage. Insert barrel shifts at outset of each outer loop traversal & initialize integer pointer variables before outer loop.

Outer Parallel Code:

1. Write wrapper around code for a single grid briquette.
   This prefetches and fetches briquette records, unpacks them, applies boundary conditions (sets ghost cells),
   performs briquette plane update, constructs new briquette record, which may be transposed, and writes back new briquette and copy, if needed, for MPI message.
   All this executes in the SPU.

2. Write wrapper around the above code.
   Based upon number of my OpenMP thread, compute which strips of briquettes to update. Call the above code to update a strip of briquettes repeatedly, keeping all 8 SPUs busy.
   Signal readiness of MPI message portion. Signal when done.
   All this executes in the SPU. PPU handles the signals.

3. All message passing, I/O, and restart dump writing in AMD.

We believe that we have now learned what works.

Now we need to lessen the programming burden.

1. Automate readable, maintainable, modifiable Fortran to Fast Fortran.
2. Automate Fast Fortran to Weird C (we have mostly done this part).
3. Parallel implementation still manual. This too could be automated.

Possible HPC Programming Model:

1. Certain subroutines and all routines they call designated for SPU.
2. Master OpenMP thread executed by PPU, which manages SPU slaves, implemented as subsidiary OpenMP threads before translation.
3. Master OpenMP thread in a team executed partially on Opteron and partially on PPU. This splitting of the master thread's tasks is peculiar to Roadrunner. Only the PPU deals with SPU thread creation and coordination. Only the Opteron executes MPI library calls. The Opteron tranmits data between its and the Cell blade's memories via DaCS library calls (peculiar to Roadrunner).
4. I/O handled by separate MPI process or processes on Opterons.
   a. This seems to be the cleanest implementation.
   b. Allows fewer than one such process per node on large systems.
   c. Consolidates and reformats data, then streams it out of system.
5. Restart dumps handled by separate MPI process on Opteron at each node, overlapped with continued code execution.

An Example Code:

It is our intent to produce and make available a fairly readable example code that illustrates the points made in this tutorial.

Such a code, with a full Roadrunner parallel implementation, is not yet available, but will be generated from a simplification of the multifluid PPM code we plan to run on the full Roadrunner configuration in June.

At the moment, the best we can do is to make available a 3-D PPM code for flows Mach 2 and below that is implemented in both slow and fast Fortran.

This is a code appropriate to run on a dual-core laptop rather than on Roadrunner, but it illustrates the most unusual features that are discussed in these slides, namely the transformation of the code destined for the Cell SPU from a slow to a fast Fortran expression.

This code contains a simplified outer code that runs the slow Fortran version in a cache-blocked mode. Alternatively, this same outer code, with a different setting of the parameters defining the size of grid sugar cubes, runs the fast Fortran version. Flop counts are reported by the code.

The Example Code:

The example code is located at www.lcse.umn.edu/RR and is to be made available only through the Los Alamos Roadrunner tutorial Web site.